Pierre-Cyrille Héam
Cyril Nicaud

Seed: an Easy-to-Use Random Generator of Recursive Data Structures for Testing

Research Report LSV-09-15

July 2009

Laboratoire Spécification et Vérification

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

SEED : an Easy-to-Use Random Generator of Recursive Data Structures for Testing

P.-C. Héam[1,3] and C. Nicaud[2]

[1] LIFC, Université de Franche-Comté & INRIA, Besançon, France
[2] LIGM, Université Paris Est & CNRS, Marne-la-Vallée, France
[3] LSV, ENS Cachan & CNRS & INRIA, Cachan, France

Abstract. Random testing represents a simple and tractable way for software assessment. This paper presents the SEED tool dedicated to the uniform random generation of recursive data structures as labelled trees or logical formulas. We show how SEED can be used in several testing contexts, from model based testing to performance testing. Generated data structures are defined by grammar-like rules, given in an XML format, multiplying SEED possible applications.
SEED is based on combinatorial techniques, and can generate uniformly at random k structures of size n with a time complexity in $O(n^2 + kn \log n)$. Finally, SEED is available as a free java application and a great effort has been made to make it easy-to-use.

1 Introduction

As software development requires costly testing phases, the automation of testing processes is a topic of considerable practical importance, leading to a large variety of methods and techniques. In this context, the random testing paradigm [1] represents a quite simple and tractable software assessment method for various testing approaches: model based testing [2,3], structural testing [4], Object-Oriented program testing [5], performance testing [6,7], *etc.* When doing random testing, the main qualities required for the random sampler are that random choices must be objective and independent of tester choices or convictions, and the algorithm has to be efficient enough to allow the generation of a huge quantity of data. For the first point, a solution is to ask for uniform (i.e. equally likely outcomes) random generators, as they have no subjective bias. Uniform random generators are also interesting, since they may allow theoretical probabilistic studies, as it is well illustrated for instance in [2,8]. Every programming languages provides very nice uniform random generators (or pseudo-random to be more precise) for numerical data. However, the question is more complex for non-numerical data, as tree data-structures, logical formulas, graphs, *etc.*

In this paper, we present the SEED[4] tool that uniformly generates recursive data structures satisfying a given grammar-like specification. Our goal is to

[4] SEED can be freely downloaded at `http://igm.univ-mlv.fr/~nicaud/seed/`

provide an easy-to-use yet expressive and efficient tool to test software for tree-like inputs, therefore including most kinds of expressions: arithmetic expressions, regular expressions, temporal logic formulas, *etc.*

The algorithmic part is based on the so-called *recursive method*, introduced by Nijenhuis and Wilf [9] and developed by Flajolet, Zimmermann and Van Cutsem in [10]. It allows to generate efficiently decomposable combinatorial structures [11] uniformly at random. Our approach is similar to the one of GENRGENS [12] that is dedicated to the generation of genomic sequences: it consists in adapting, specializing and implementing general algorithms from combinatorics to another area of research where random generators are needed.

As exposed in [13], random testing is frequently criticized for two reasons: first, random generation is not easy on non numerical data and, second, random testing is neither selective nor directed: if *error-cases* are very infrequent, they are not likely to be discovered. The tool presented in this paper answers this two points: first, we will show that random generation of complex inputs is possible with SEED and, second, that SEED features make it suitable to be combined with other testing techniques as model-based testing or combinatorial testing. Random testing may be either directed or selective. We also expose how SEED can be particularly useful for performance testing algorithms.

1.1 A First Grasp

Consider a software manipulating regular expressions. In order to evaluate its performances, suppose that one wants to compute the average running time for 1000 random inputs of size 100. On a two-letters alphabet $A = \{a, b\}$, a regular expression is made of the letters in A, the empty word λ, combined using unions, concatenations and star operations. For instance $a \cdot (a \cdot b)^* \cup \lambda$ is a regular expression. The set E of tree representations of regular expressions can be informally defined with the rule:

$$E := \lambda \mid a \mid b \mid \overset{*}{\underset{E}{|}} \mid \overset{\bullet}{\underset{E\ E}{/\backslash}} \mid \overset{\cup}{\underset{E\ E}{/\backslash}}$$

Translating directly this description in a specific XML format, one can use SEED to obtain the 1000 random regular expressions, stored as an XML file too, using a command such as

```
java -jar seed.jar RegExp.xml 100 1000 > random_data.xml
```

This only takes a few seconds.

The random generation is uniform, according to the specification. Note that in the specification above, $a \cup b$ and $b \cup a$ are considered as two distinct expressions and terms such as $a \cup a$ can be generated. This may introduce a bias in the distribution that one want to avoid. As commutative or idempotent operators often appear in practice, SEED has been designed to handle them simply: changing a tag `<binary id="Union">` into `<binary_sym_neq id="Union">` is all one has to do for SEED to consider $E_1 \cup E_2$ and $E_2 \cup E_1$ as the same expression, and to forbid any pattern of the form $E \cup E$. Moreover, the random generation is still uniform with this new rules added.

1.2 Application to Testing

Here are some testing problems where random generators can be useful:

- Reliability Testing: One wants to Black-box testing a given software to ensure that exceptions are handled properly. For instance, if the software uses regular expressions as inputs, and should return an exception if the expression contains the pattern $\overset{\cup}{\underset{* *}{/\backslash}}$, one can use the random generator of the previous section to perform the tests. If one want to insist on correct inputs, it is possible to change the specification in order to generate correct regular expressions only:

$$\begin{cases} E := X \mid S \\ S := \overset{*}{\underset{X}{|}} \\ X := \lambda \mid a \mid b \mid \overset{\bullet}{\underset{E\ E}{/\backslash}} \mid \overset{\cup}{\underset{X\ X}{/\backslash}} \mid \overset{\cup}{\underset{X\ S}{/\backslash}} \mid \overset{\cup}{\underset{S\ X}{/\backslash}} \end{cases}$$

  Such a system of equations can be handled by SEED , and using E as the main class ensures that each generated expression avoid the forbidden pattern.
- Performance Testing: One want to compare the average performances, for various criteria, of several tools performing the same task. Note that in this case also, the possibilities of SEED make it easy to generate fragments of the set of inputs. It can be useful in order to focus on some particular structures, typically containing or avoiding a tree pattern, while guaranteeing the uniformity.
- Symbolic Testing: The approach is based on a finite symbolic partition of data-inputs. Two data in the same class are considered as equivalent. The goal is then to generate classes of equivalences, by generating witnesses. If a class is particularly small, random testing based on an uniform distribution on data will miss it with high probability. SEED proposes interesting features to try tackle this problem, especially adapted for expressions: First, as shown above, if classes of the partition can be expressed by a system of equations, it is possible to pick uniformly at random a witness of the classes. Moreover, several semantic equivalences has been implemented in SEED, as has been shown in the first simple example. Operators can be defined either as commutative or idempotent or associative, providing a quite better (semantically) random generation for our specific problems.

1.3 Contributions

SEED is Java tool provided by a .jar file that allows the uniform random generation of tree-like data structures defined by grammar-like specifications. More precisely, SEED provides following features:

- SEED generates finite trees whose nodes may be labelled either by binaries operators (which may possibly be either commutative or idempotent or both commutative and idempotent), or by unary operators, or by constant symbols or by an unbounded in arity operator (sequences).

- Generated trees are in a sub-algebra of labelled trees which is specified using a grammar-like system.
- Trees are generated uniformly at random, for any given size n, with respect to operators properties (i.e. commutativity). Moreover, the time complexity to generate k structures is in $O(n^2 + kn \log n)$: the generation is efficient and quite large objects can be generated.
- SEED allows to count the number of objects of a given size satisfying the specification.
- Both specifications and results are encoded in readable and easily parsable XML files, making SEED easy-to-use.
- SEED can also generate closed first order formulas in prenex form, by randomly generating the quantifiers (formulas are generated as trees). It is also possible to generate formulas in some first order fragments classically defined by quantifier alternations. Generation is still uniform for logical formulas.

SEED in has been developed for testing purposes and to be easy-to-use. SEED may be useful for validation engineers in a software testing context (several examples will be provided), for developers for debugging, for researcher to evaluate or test conjectures and for tool-users to compare performances. XML input/output files and command line use of SEED make it particularly easy to manipulate with command scripts.

1.4 Related Work

There are three main methods to generate uniformly at random combinatorial structures: the recursive method [10], Boltzmann samplers [14] and techniques using random walks in Markov chains [15]. SEED is based on the recursive method, as are other tools, for instance GENRGENS[5] [12] and the CS package of MUPAD [16]. GENRGENS is a tool dedicated to bioinformatics, easy to use, which can generate context-free classes of objects. CS uses all the possibilities of the recursive method but is mostly dedicated to combinatorists and thus require some knowledge to be used efficiently.

In fact SEED is a restriction of this general method, where the constructions that are likely to be useful when describing tree-like structures have been selected. It is more expressive than genRgenS, which can not describe constructions as commutative operators, and less expressive than CS which implements the general theory, at the cost of being less user-friendly for specific areas of application.

In order to generate random structures for testing defined by context-free grammars, [17,18] approaches are similar to ours. However, neither symmetry and impotency of operators nor sequences are handled. Furthermore there is no available implementation and time complexity is not optimal (generating k recursive structures of size n requires $O(kn^2)$ operations while SEED requires $O(n^2 + kn \log n)$ operations). Note that these algorithms were used in [19] for

[5] `http://www.lri.fr/~genrgens/`

testing in a black-box fuzzing context. In [20] a random approach from grammar based specifications is proposed. However, involved recursive constructions are less expressive than SEED's ones and random generation procedure is not uniform.

Other non-random grammar based approach are used for testing. Work [21] addresses parser testing and [22] focuses on testing re-factoring engines (program transformation software). A systematic generation for grammar based inputs is proposed in [23]. However because of the explosion of tests, symbolic approaches are preferred as in [24,13,19].

1.5 Layout of the Paper

SEED is based on a formal background described in Section 2. The presentation of SEED is done in Section 3. Several examples of applications are developed in Section 4. Finally, Section 5 concludes and gives some perspectives.

2 Theoretical Aspects

In order to precisely and rigorously expose SEED features, the theoretical background presented in this section is needed.

Lets consider a finite set of labels $\mathcal{F}$ which is a disjoint union of finite sets $\mathcal{F}_0^{\text{var}}$, $\mathcal{F}_0$, $\mathcal{F}_1$, $\mathcal{F}_2$, $\mathcal{F}_2^{\text{sym}}$, $\mathcal{F}_2^{\text{neq}}$, $\mathcal{F}_2^{\text{sym}-\text{neq}}$ and $\mathcal{F}^{\text{seq}}$ respectively called *set of variables*, *set of constants*, *set of unary operators*, *set of binary operators*, *set of binary-symmetric operators*, *set of binary-idempotent operators*, *set of binary-symmetric-idempotent operators* and *set of sequential operators*.

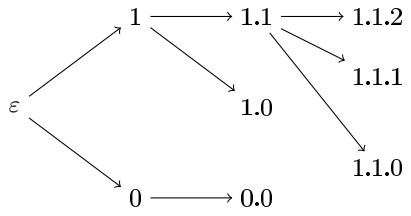
2.1 Labelled Trees

A *tree* t is a finite prefix-closed subset of $\mathbb{N}^*$ (words over integers) such that if $p \cdot i \in t$, with $i \in \mathbb{N}$, then for every $j \leq i$, $p \cdot j \in t$. Elements of t are called *nodes*. An example is depicted in Fig 1. The element ε -the empty word- is the *root* of the tree. Then nodes are recursively numbered: if p is in t, the element $p \cdot i$ in t ($i \in \mathbb{N}$) is the i-th child of p. Maximal elements of t for the prefix order are the *leaves* of t.

If t is a tree, the i-th subtree of t, for $i \in \mathbb{N}$ is the tree $i^{-1} \cdot t = \{u \in \mathbb{N}^* \mid i \cdot u \in t\}$. It coincides with the intuitive notion of subtree. Note that $i^{-1} \cdot t$ can be empty.

A *labelled tree* on $\mathcal{F}$ is a pair (t, φ), where t is a tree and φ a function from t into $\mathcal{F} \cup 2^{\mathcal{F}_0}$ satisfying:

- $\varphi(p) \in \mathcal{F}_0^{\text{var}} \cup \mathcal{F}_0 \cup 2^{\mathcal{F}_0}$ if and only if p is a leaf (leaves are labelled by variables, constants or sets of constants).
- if $\varphi(p) \in \mathcal{F}_2 \cup \mathcal{F}_2^{\text{neq}} \cup \mathcal{F}_2^{\text{sym}-\text{neq}} \cup \mathcal{F}_2^{\text{sym}}$, then $p \cdot 0 \in t$ and $p \cdot 1 \in t$ and $p \cdot 2 \notin t$ (p is a binary node).



The tree $t = \{\varepsilon, 0, 1, 0 \cdot 0, 1 \cdot 0$
$1 \cdot 1, 1 \cdot 1 \cdot 0, 1 \cdot 1 \cdot 1, 1 \cdot 1 \cdot 2\}$

Fig. 1. A tree

- if $\varphi(p) \in F_2^{\text{neq}} \cup \mathcal{F}_2^{\text{sym}-\text{neq}}$, then there exists $q \in \mathbb{N}$ such that either $p \cdot 1 \cdot q \in t$ but $p \cdot 0 \cdot q \notin t$, or $p \cdot 0 \cdot q \in t$ but $p \cdot 1 \cdot q \notin t$, or $\varphi(p \cdot 1 \cdot q) \neq \varphi(p \cdot 0.pq)$ (the subtrees of p are different).
- if $\varphi(p) \in \mathcal{F}_1$, then $p \cdot 0 \in t$ and $p \cdot 1 \notin t$ (p is a unary node).

Incidentally, each node that is not a leaf can be labelled by an element of $\mathcal{F}^{\text{seq}}$, and a node who has three or more children is necessarily labelled by an element in $\mathcal{F}^{\text{seq}}$. For instance, a labelled tree (t, φ) for the tree depicted in Fig. 1 has to satisfy $\varphi(\varepsilon) \in \mathcal{F}_2 \cup \mathcal{F}_2^{\text{neq}} \cup \mathcal{F}_2^{\text{sym}-\text{neq}} \cup \mathcal{F}_2^{\text{sym}} \cup \mathcal{F}^{\text{seq}}$, $\varphi(1) \in \mathcal{F}_2 \cup \mathcal{F}_2^{\text{neq}} \cup \mathcal{F}_2^{\text{sym}-\text{neq}} \cup \mathcal{F}_2^{\text{sym}} \cup \mathcal{F}^{\text{seq}}$, $\varphi(0) \in \mathcal{F}_1$, $\varphi(1 \cdot 1) \in \mathcal{F}^{\text{seq}}$, and for other elements p of t, $\varphi(p) \in \mathcal{F}_0^{\text{var}} \cup \mathcal{F}_0 \cup 2^{\mathcal{F}_0}$.

If $T = (t, \varphi)$ is a labelled tree, its i-th *labelled subtree* (or *subtree* for short if it is clear in the context that the trees are labelled) is the tree $i^{-1} \cdot T = (s, \psi)$ such that $s = i^{-1} \cdot t$ and for all $u \in s$, $\psi(u) = \phi(i \cdot u)$.

The *size* of a node p of a labelled tree is 1 if $\varphi(p) \notin 2^{\mathcal{F}_0}$ and $|\varphi(p)|$ if $\varphi(p) \in 2^{\mathcal{F}_0}$. The size of a tree is the sum of the sizes of its elements. Informally, the size of a tree is equal to the number of its nodes where leaves labelled by sets are counted as many times as the cardinal of their label. For instance in Fig. 1 if leaves $1 \cdot 1 \cdot 0$ and $1 \cdot 1 \cdot 1$ are labelled by elements in $\mathcal{F}_0$, and if $1 \cdot 1 \cdot 2$ is labelled by a 5 element subset of $\mathcal{F}_0$, then the size of the labelled tree is 12 (7 nodes of size 1 and a node of size 5).

2.2 Labelled Tree Isomorphisms

It remains to define the notion of labelled tree isomorphism. The isomorphism relation $\sim$ on labelled trees is defined inductively for $T_1 = (t_1, \varphi_1)$ and $T_2 = (t_2, \varphi_2)$ by:

- If both t_1 and t_2 are empty, they are isomorphic.
- Otherwise, it is necessary that $\varphi_1(\varepsilon) = \varphi_2(\varepsilon)$ and that
  - If $\varphi_1(\varepsilon) \notin \mathcal{F}_2^{\text{neq}} \cup \mathcal{F}_2^{\text{sym}-\text{neq}}$, $i^{-1} \cdot T_1$ and $i^{-1} \cdot T_2$ are isomorphic for any $i \in \mathbb{N}$.
  - If $\varphi_1(\varepsilon) \in \mathcal{F}_2^{\text{neq}} \cup \mathcal{F}_2^{\text{sym}-\text{neq}}$, either $0^{-1} \cdot T_1 \sim 0^{-1} \cdot T_2$ and $1^{-1} \cdot T_1 \sim 1^{-1} \cdot T_2$, or $1^{-1} \cdot T_1 \sim 0^{-1} \cdot T_2$ and $0^{-1} \cdot T_1 \sim 1^{-1} \cdot T_2$. I.e. their children are isomorphic, up to permutation.

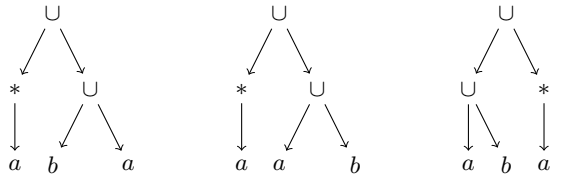


Fig. 2. Isomorphic Labelled Trees

For instance, if $\cup \in \mathcal{F}_2^{\mathrm{sym-neq}}$, the three labelled trees depicted in Fig. 2 are isomorphic: one can transform the first one into the second one by permuting the two leaves a and b, and from the second one into the third one by permuting the two children of the root.

2.3 Grammar Defined Labelled Tree languages

Given a set of *grammar symbols* Σ (we assume that $\Sigma \cap \mathcal{F} = \emptyset$), a *rule-expression* is inductively defined by:

- X is a rule-expression, for every $X \in \Sigma \cup \mathcal{F}_0 \cup \mathcal{F}_0^{\mathrm{var}}$,
- if $E_1, \ldots, E_k$ are rule-expressions, so is $E_1 | \ldots | E_k$,
- $\mathrm{Seq_f(X)}$ is a rule-expression for every $X \in \Sigma$ and $f \in \mathcal{F}^{\mathrm{seq}}$,
- XfY is a rule-expression for every $X, Y \in \Sigma$ and $f \in \mathcal{F}_2$, (sometimes, the expression XfY is denoted $f(X, Y)$),
- XfX is a rule-expression for every $X \in \Sigma$ and $f \in \mathcal{F}_2^{\mathrm{neq}} \cup \mathcal{F}_2^{\mathrm{sym-neq}} \cup \mathcal{F}_2^{\mathrm{sym}}$, (sometimes, the expression XfY is denoted $f(X, Y)$),
- fX is a rule-expression for every $X \in \Sigma$ and $f \in \mathcal{F}_1$, (sometimes, the expression fX is also denoted $f(X)$ or Xf),
- $\mathrm{Set}[c_1, \ldots, c_k]$ is a rule-expression for every $\{c_1, \ldots, c_k\} \subseteq \mathcal{F}_0$,
- $\mathrm{Set}_j[c_1, \ldots, c_k]$ is a rule-expression for every $\{c_1, \ldots, c_k\} \subseteq \mathcal{F}_0$ and $j \in \mathbb{N}$.

Example: $a \mid b \mid \lambda \mid \mathrm{star}(H) \mid G \cup G \mid G \cdot G$, with $\mathcal{F}_0 = \{a, b, \lambda\}$, $\mathcal{F}_1 = \{\mathrm{star}\}$, $\mathcal{F}_2^{\mathrm{sym-neq}} = \{\cup\}$, $\mathcal{F}_2 = \{\cdot\}$ and $G, H \in \Sigma$ is a rule-expression.

A *grammar* is a pair $\mathcal{G} = (A, \mathcal{R})$, where $A \in \Sigma$ is the initial symbol and $\mathcal{R}$ is a partial function from Σ into the set of rule expressions. An element (Z, E) of $\mathcal{R}$ is denoted $Z := E$.

Example: $\mathcal{G}_{\mathrm{exe}} = (G, \{G := a \mid b \mid \lambda \mid \mathrm{star}(H) \mid G \cup G \mid G \cdot G, H := a \mid b \mid \lambda \mid G \cup G \mid G \cdot G\})$ is a grammar.

For every $Z \in \Sigma$ and rule expression E we inductively define the set of labelled trees $L_{\mathcal{G}}(Z)$ and $L_{\mathcal{G}}(E)$ by: $L_{\mathcal{G}}(Z) = \emptyset$ if Z is not in the domain of $\mathcal{R}$, and if $Z := E$, then $L_{\mathcal{G}}(Z) = L_{\mathcal{G}}(E)$ where

- if $E = X \in \Sigma$, then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $(t, \varphi) \in L_{\mathcal{G}}(X)$,
- if $E \in \mathcal{F}_0 \cup \mathcal{F}_0^{\mathrm{var}}$, $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon) = E$,
- if $E = E_1 | \ldots | E_k$, then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $(t, \varphi) \in \bigcup_i L_{\mathcal{G}}(E_i)$

- if $E = \mathrm{Seq_f(X)}$, then $T = (t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon) = f$ and for every $i \in \mathbb{N}$, the labelled subtree $i^{-1} \cdot T$ is in $L_{\mathcal{G}}(X)$,
- if $E = f(X)$, then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon) = f$ and the labelled subtree $0^{-1} \cdot T$ is in $L_{\mathcal{G}}(X)$,
- if $E = XfY$, then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon) = f$ and the labelled subtrees $0^{-1} \cdot T$ and $1^{-1} \cdot T$ are respectively in $L_{\mathcal{G}}(X)$ and $L_{\mathcal{G}}(Y)$,
- if $E = XfX$, then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon) = f$ and the labelled subtrees $0^{-1} \cdot T$ and $1^{-1} \cdot T$ are both in $L_{\mathcal{G}}(X)$,
- if $E = \mathrm{Set}[c_1, \ldots, c_k]$ then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon) \subset \{c_1, \ldots, c_k\}$
- if $E = \mathrm{Set}_j[c_1, \ldots, c_k]$ then $(t, \varphi) \in L_{\mathcal{G}}(E)$ iff $\varphi(\varepsilon)$ is a subset of $\{c_1, \ldots, c_k\}$ of cardinal j.

The *language* accepted by a grammar is the language defined by its initial symbol. For instance for the grammar $\mathcal{G}_{\mathrm{exe}}$, $L_{\mathcal{G}_{\mathrm{exe}}}(H)$ is the set of labelled trees whose root is not labelled by $*$, and $L_{\mathcal{G}_{\mathrm{exe}}}(G)$ is the set of labelled trees satisfying: the child of a node labelled by $*$ is not labelled by $*$.

2.4 Random Generation

Given a grammar $\mathcal{G}$, positive integers n, k, SEED generates k labelled trees of the language accepted by $\mathcal{G}$. The uniformity is defined up to isomorphism of labelled trees.

As mentioned above, SEED uses a reduced set of operators of the recursive method [10]. The random generation is done in two steps (see Fig. 3): a precalculus of the number of objects of size in $\{1, \cdots, n\}$ and the random generations thereafter. The precalculus is of quadratic complexity, and each random generation is done in $O(n \log n)$ time, using a boustrophedon's method (see [10] for the details). The overall complexity is therefore $O(n^2 + kn \log n)$. Remark that it is more efficient to generate k random trees once than a random tree k times, because of the precalculus.

3 Tool Description

3.1 Tool Presentation

SEED is a java tool provided by the `seed.jar` file. The architecture of the tool is depicted in Fig 3 and follows the theoretical approach described in Section 2.4. The tool SEED takes as inputs a specification file of the grammar, a size n, together with an optional number k of tests and generates an XML file containing the k labelled trees of size n on the standard output: SEED is easy to using pipes or redirections. Running SEED can be easily done by the command line

```
java -jar seed.jar spec.xml n k
```

where `spec.xml` is the specification file, `n` is the size of generated trees and `k` is the number of generated trees. Several examples of specification files are provided in SEED web page.

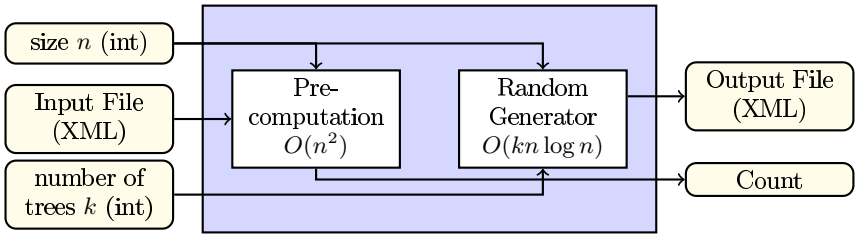


Fig. 3. Seed Architecture

It is also possible to count the number of objects of size n by changing in the above command line `k` by `-count`. Moreover, for debugging purposes, it is possible to have a text output, by replacing `k` by `-text` in the command line (only one element can be generated this way).

3.2 Input/Output Files

For a lack of room, one cannot explicitly describe XML specification files. We give a part of the specification for the one rule grammar:

$$E := a \mid b \mid \lambda \mid \mathrm{star}(E) \mid E \cup E \mid E \cdot E,$$

and $\mathcal{F}_0 = \{a, b\}$, $\mathcal{F}_1 = \{\mathrm{star}\}$, $\mathcal{F}_2^{\mathrm{sym-neq}} = \{\cup\}$, $\mathcal{F}_2 = \{\cdot\}$ (other undefined symbols sets are empty) defining regular expressions. For instance the parts $E \cup E$ and $\mathrm{star}(E)$ would be encoded by:

```
<binary_sym_neq id="Union">
   <symbol value="U"/>
   <child idref="E"/>
</binary_sym_neq>
```

```
<unary id="Star">
   <symbol value="star"/>
   <child idref="E"/>
</unary>
```

The Union operator is denoted `U` (`<symbol value="U"/>`), it is in $\mathcal{F}_2^{\mathrm{sym-neq}}$ (`<binary_sym_neq id="Union">... </binary_sym_neq>`), and its children are generated by the symbol `E`. (`<child idref="E"/>`). A precise description and a tutorial are available on SEED web page.

Output files are very readable. Each generated tree is marked by the `<tree>` and `</tree>` tags. The structure of the generated trees exactly follows the tree structure of the XML output.

4 Application Examples

4.1 Simple Illustration Examples

One can define polynomial functions on $\mathbb{N}$ using Peano Arithmetic. This can be performed using the unique rule:

$$P := 0 \mid x \mid s(P) \mid P + P \mid P * P,$$

and $\mathcal{F}_0 = \{0, x\}$, $\mathcal{F}_1 = \{s\}$, $\mathcal{F}_2^{\text{sym}} = \{+, *\}$,(other undefined symbols sets are empty). Symbols $+$ and $*$ are encoded as symmetric operators since addition and multiplication are commutative.

Generating 3SAT formulas over atomic propositions $a_1, \ldots, a_k$ is easy with SEED using the following grammar (initial symbol is F):

$$F := \text{Seq}_{\wedge}(C) \quad \text{and} \quad C := \text{Set}_3[A] \quad \text{and} \quad A := a_1 | \ldots | a_k | \neg a_1 | \ldots | \neg a_k.$$

Symbol C encodes clauses, A encodes finite set of atomic propositions and their negations. The size of a generated tree is four times the number of clauses plus one: random generation based on the number of clauses is therefore possible. Changing $C := \text{Set}_3[A]$ by $C := \text{Set}[A]$ in the specification would generate logical Boolean formulas in conjunctive normal form.

We describe now a first order logical specification. Presburger Logic is the first order logic over integers using addition and equality. It is widely used for model-checking purposes (see e.g. [25]). Arithmetic expressions using variables $x_1, \ldots, x_4$ are defined by:

$$A := x_1 \mid x_2 \mid x_3 \mid x_4 \mid 0 \mid 1 \mid -1 \mid A + A \mid A = A,$$

and closed formulas by $P := A \mid P \vee P \mid \neg P$, where $\mathcal{F}_0 = \{0, 1, -1\}$, $\mathcal{F}_0^{\text{var}} = \{x_1, \ldots, x_4\}$, $\mathcal{F}_1 = \{\neg\}$, $\mathcal{F}_2^{\text{sym}} = \{+\}$ and $\mathcal{F}_2^{\text{sym-neq}} = \{=, \vee\}$. Using appropriate tags in the specification, it is for instance possible to generate first order formulas of the $\exists^*\forall^*\varphi$ fragment. Random generation of such kinds of fragments may be useful for instance for data bases applications [26].

4.2 Performance Testing: LTL formulas Reductions

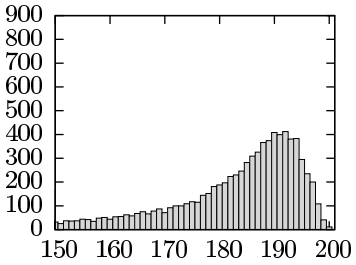
Linear Temporal Logic [27] is used in thousands of designing, verification or testing works. Depending on temporal used operators, several definitions (semantically equivalent) are used. Many tools and works (see e.g. [28,29,30]) are dedicated to the LTL-model checking. Theses tools are frequently decomposed into three parts: first the LTL formula is simplified, next it is transformed into an equivalent Büchi Automata and, to finish, model-checking is performed using classical automata constructions. The used syntax of LTL formula (over constants a, b) for model-checker is given by the following grammar:

$$L := a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \diamond L \mid \Box L \mid LUL \mid LRL.$$

We will consider two specifications for LTL formulas. The first one, Sp1, uses this unique rule with $\mathcal{F}_0 = \{a, b, \neg a, \neg b\}$, $\mathcal{F}_1 = \{\circ, \diamond, \Box\}$, $\mathcal{F}_2 = \{\vee, \wedge, U, R\}$. The second specification Sp2 considers that U and R operators are idempotent and that $\vee$ and $\wedge$ operators are symmetric and idempotent. Moreover, formulas can not have two consecutive $\Box$ symbols or two consecutive $\diamond$ symbols. This is done using the following grammar:

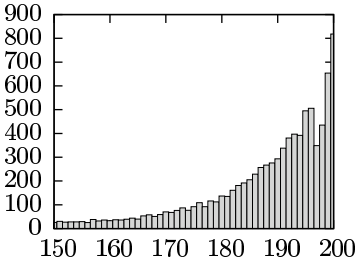
$$\begin{aligned}
L &:= a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \diamond D \mid \Box S \mid LUL \mid LRL \\
D &:= a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \Box S \mid LUL \mid LRL \\
S &:= a \mid b \mid \neg a \mid \neg b \mid L \vee L \mid L \wedge L \mid \circ L \mid \diamond D \mid LUL \mid LRL
\end{aligned}$$

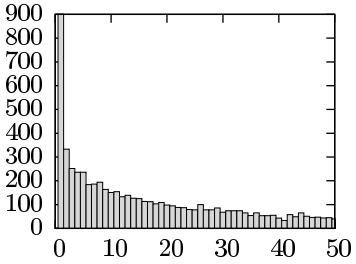
with $\mathcal{F}_0 = \{a, b, \neg a, \neg b\}$, $\mathcal{F}_1 = \{\circ, \diamond, \Box\}$, $\mathcal{F}_2^{\text{neq}} = \{U, R\}$ and $\mathcal{F}_2^{\text{sym}-\text{neq}} = \{\vee, \wedge\}$.



For the simple specification Sp1, simplified trees have ≈ 39.1 less nodes in average, even if the most probable size of a resulting simplified tree is around 190. Note that the probability of generating an irreducible tree is very small: one can not use a rejection algorithm for this purpose efficiently. It also means that the resulting distribution after simplification has a serious bias.

For the specification Sp2, simplified trees have ≈ 24.9 less nodes in average: there are less useless parts in the generated trees. The probability of generating an irreducible tree is about 8.2%. It is still small, but at the costs of multiplying the average generation time by 12, one can use a rejection algorithm to obtain a uniform distribution on irreducible trees.





For this diagram, $\top$ and $\bot$ has been added as constants to the simple specification. As could have been expected, though it may seems natural to add them, the simplified trees have been dramatically reduced. The diagram has been truncated: more than 30% of the simplified trees are trivial.

Fig. 4. Distribution of sizes after applying simplification rules on LTL formulas, generate with three different grammars. For each diagram, 1000 LTL formulas of size 200 has been randomly generated. The size of the simplified trees are on the x-axis, the height of the column represents the number of hits, out of the 1000 generations, for each given size. Note that the range value has been cut, to focus on the most significant parts.

The study of the efficiently of rewriting rules on formulas generated by the two specifications is depicted in Fig. 4. We used the following simplification rules [29] depicted in Fig. 5.

4.3 Combinatorial Testing

We show how our approach can be used in a scenario based combinatorial testing approach (see e.g. [31,32]). In this context, test cases are sequences of actions

$E \vee E \rightarrow E$ | $E \wedge E \rightarrow E$ | $E \vee \neg E \rightarrow \bot$ | $E \wedge \neg E \rightarrow \top$
$E \vee \top \rightarrow \top$ | $E \vee \bot \rightarrow E$ | $E \wedge \bot \rightarrow \bot$ | $E \wedge \top \rightarrow E$
$\Box\Box E \rightarrow \Box E$ | $\Diamond\Diamond E \rightarrow \Diamond E$ | $\Diamond\top \rightarrow \top$ | $\Diamond\bot \rightarrow \bot$
$EU\top \rightarrow \top$ | $EU\bot \rightarrow \bot$ | $\top U E \rightarrow \Diamond E$ | $\bot U E \rightarrow E$
$ER\top \rightarrow \top$ | $ER\bot \rightarrow \bot$ | $\top R E \rightarrow E$ | $\bot R E \rightarrow \Box E$
$ER\top \rightarrow \top$ | $ER\bot \rightarrow \bot$ | $\top R E \rightarrow E$ | $\bot R E \rightarrow \Box E$
$(\circ E)U(\circ F) \rightarrow \circ(EUF)$ | $\circ\top \rightarrow \top$ | $(\circ E) \wedge (\circ F) \rightarrow \circ E \wedge F$
$\Box \Diamond E \vee \Box \Diamond F \rightarrow \Box \Diamond (F \vee E)$ | | $EU\Box \Diamond F \rightarrow \Box \Diamond F$
$\Diamond \circ E \rightarrow \circ \Diamond E$ | | $\circ\Box \Diamond E \rightarrow \Box \Diamond E$
$\Diamond(E \wedge \Box \Diamond F) \rightarrow (\Diamond E) \wedge (\Box \Diamond F)$ | | $\Box(E \vee \Box \Diamond F) \rightarrow (\Box E) \vee (\Box \Diamond F)$
$\circ(E \wedge \Box \Diamond F) \rightarrow (\circ E) \wedge (\Box \Diamond F)$ | | $\circ(E \vee \Box \Diamond F) \rightarrow (\circ E) \vee (\Box \Diamond F)$

Fig. 5. LTL formulas Simplification Rules

(with values) that are chosen according to a scenario defined by a kind of regular expression. For a lack of room and because it is not the purpose of this paper, we show how to use our tool on a fragment of [31], extension to the general case should be obvious. Given a set Σ of actions, where X is a finite set of values, a *scenario* is an expression inductively defined by: $a(Y)$ with $a \in \Sigma$ and $Y \subseteq X$ is a scenario and if E_1, E_2 are scenario, $E_1 + E_2$, $E_1.E_2$, E_1^*, $E_1^{k_1,k_2}$, with $k_1, k_2 \in \mathbb{N}$ $(k_1 \leq k_2)$ are scenarios. The language $L(E)$ accepted by a scenario E is a language of $(\Sigma \times 2^X)^*$ defined by: $L(a(Y)) = \{a(y) \mid y \in Y\}$ and $L(E_1 + E_2) = L(E_1) \cup L(E_2)$, $L(E_1.E_2) = L(E_1).L(E_2)$ (concatenation), $L(E_1^*) = L(E_1)^*$ (Kleene star), $L(E_1^{k_1,k_2}) = L(E_1)^{k_1}. \cup_{0 \leq i \leq k_2 - k_1} L(E_1)^i$. For instance $L(a(\{5,9\})^{2,3}) = \{a(i)a(j), a(i)a(j)a(k) \mid i,j,k \in \{5,9\}\}$.

Using SEED , given a scenario E and an integer n, one can uniformly generate test cases of length n. It suffices to encode the regular expression into a grammar. Consider for instance the scenario $E = a(\{3,4\})^{1,5}.(b(\{2,5\}) + c(\{1\})^*$. One can encode it with the grammar (initial symbol is S):

$$A_1 := a(3) \mid a(4) \qquad B := b(5) \mid b(2) \mid c(1)$$
$$A_2 := A.A \qquad P_2 := B \mid B.P_2$$
$$A_4 := A_2.A_2 \qquad S := P_1 \mid P_1.P_2$$
$$P_1 := A_1 \mid A_2 \mid A_1.A_2 \mid A_4 \mid A_1.A_4$$

Symbol P_1 recognizes $a(\{3,4\})^{1,5}$, symbol P_2 recognizes $(b(\{2,5\}) + c(\{1\})^+$.

4.4 Model Based Testing

Several works [33,34,35] are dedicated to the model-based testing of systems using logical/algebraic specifications: the SEED feature for generating first order formulas would be particularly interesting for this purpose. In [2] authors shows how to informally generate paths of a given length in finite state machines and how to use this generation in a model-based testing approach. We show now how to use SEED allows to extend the approach [2] to tree executions.

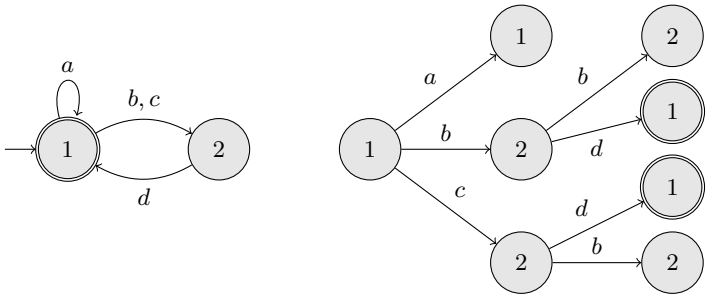


Fig. 6. Random-Generation for Model-based Testing

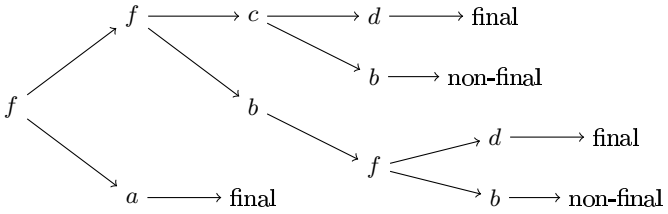


Fig. 7. Test-case Encoding

In [36,37] test objectives are defined by an Input-Output Labelled Transition System, i.e. a kind of finite automaton and test cases are spanning trees of unfolding of the transition systems. For a lack of room, we don't precisely defined theses notions and just explain how to use SEED in this context on a simplified example described by a finite automaton. Generalization should be quite easy and would be easily automatized. Fig 6 depicts the specification of a system by a finite automaton with two states and a test-case.

A test case is an execution tree of the automaton such that for each non leaf node of the tree, all transitions from this node are covered in the tree, i.e. the execution tree is complete. Moreover, leaves are marked as final or non final, depending on the corresponding state (in the context of IOLST-based testing, the good denomination is *pass* or *fail*). We use a binary encoding of this tree, using a symbol f to encode sequences of children. We also encode labels of transitions into nodes. For instance the test-case depicted in Fig. 6 is described in Fig. 7.

This can be performed using the following rules, S_1 being the initial symbol:

$$\begin{array}{ll} S_1^a := a(S_1) & S_b^2 := b(S_2) \\ S_1 := f(S_a^1, S_1') \mid \text{final} & S_c^2 := c(S_2) \\ S_1' := f(S_b^2, S_c^2) & S_d^1 := d(S_1) \\ S_2 := f(S_b^2, S_c^2) \mid \text{not} - \text{final} & \end{array}$$

Of course, binary encoding change the distribution. However, if the starting automaton is complete, the distribution remains uniform.

4.5 Application to Combinatorics

It is not the main purpose of SEED, but it can be used to assist combinatorists working on tree structures. In this context the `-count` option should be useful to identify the associated sequences. For instance, the complete binary trees, counted by Catalan numbers are described by

$$B := \bullet \mid \overset{\bullet}{\underset{B\ B}{\wedge}},$$

with the binary operator in $\mathcal{F}_2$.

The unary-binary trees, counted by Motzkin numbers, are described by

$$U := \bullet \mid \overset{\bullet}{\underset{U}{|}} \mid \overset{\bullet}{\underset{U\ U}{\wedge}},$$

with the binary operator in $\mathcal{F}_2$.

If the binary operator of the previous grammar rule is in $\mathcal{F}_2^{\text{sym}}$, one obtain the rooted non-plane unary binary trees, counted by the Wedderburn-Etherington numbers.

5 Conclusion and Related Work

In this paper we have introduced the SEED tool dedicated to the random generation of recursive data structures. Using efficient algorithms from combinatorics, SEED can generate a lot of different kinds of structures. This structures can be easily described by the user using an XML file.

As tree structures encode expressions and formulas, SEED can been use to generate random inputs for a many applications, a few of them have been briefly presented in this article.

The next new features we intend to implement in SEED are:

- The option to use floating point approximations in the precalculus instead of big integers. In this version of SEED, the size of the generated is limited to about 400 to 500, depending on the specification. This is due to memory limitation, as one has to stock a lot of very big integers. Floating point approximations would allow the generation of bigger structures when needed, at the cost of a small bias in uniformity.
- The possibility to set the size of operators to zero, that could be useful in some situations.
- Include in the input grammar the possibility of specifying forbidden patterns or mandatory patterns. The idea here is not to extend the expressivity of SEED, but to simplify the description of complex fragments in the specification. Note that this feature requires a new theoretical study to guarantee the efficiency of the algorithm.

References

1. Hamlet, D.: Random Testing. Encyclopedia of Software Engineering. John Wiley and Sons (1994)
2. Gaudel, M.C., Denise, A., Gouraud, S.D., Lassaigne, R., Oudinet, J., Peyronnet, S.: Coverage-biased random exploration of models. Electr. Notes Theor. Comput. Sci. **220**(1) (2008) 3–14
3. Sen, K.: Effective random testing of concurrent programs. [38] 323–332
4. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In Sarkar, V., Hall, M.W., eds.: PLDI, ACM (2005) 213–223
5. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ICSE, IEEE Computer Society (2007) 75–84
6. Tauriainen, H.: Automated testing of Büchi automata translators for linear temporal logic. Research Report A66, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland (December 2000)
7. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In Sutcliffe, G., Voronkov, A., eds.: LPAR. Volume 3835 of Lecture Notes in Computer Science., Springer (2005) 396–411
8. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Trans. Software Eng. **10**(4) (1984) 438–444
9. Nijenhuis, A., Wilf, H.S.: Combinatorial Algorithms. Academic Press (1978)
10. Flajolet, P., Zimmermann, P., Cutsem, B.V.: A calculus for the random generation of labelled combinatorial structures. TCS **132**(2) (1994) 1–35
11. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2008)
12. Ponty, Y., Termier, M., Denise, A.: Genrgens: Software for generating random genomic sequences and structures. Bioinformatics **22**(12) (2006) 1534–1535
13. Majumdar, R., Xu, R.G.: Directed test generation using symbolic grammars. [38] 134–143
14. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann samplers for the random generation of combinatorial structures. Combinatorics, Probability & Computing **13**(4-5) (2004) 577–625
15. Propp, J.G., Wilson, D.B.: Exact sampling with coupled Markov chains and applications to statistical mechanics. Random Structures and Algorithms **9**(1&2) (1996) 223–252
16. Denise, A., Dutour, I., Zimmermann, P.: Cs: a mupad package for counting and randomly generating combinatorial structures. In: FPSAC'98. (1998) 195–204
17. McKenzie, B.: Generating string at random from a context-free grammar. Technical Report TR-COSC 10/97, Univ ersity of Canterbury (1997)
18. Hickey, T.J., Cohen, J.: Uniform random generation of strings in a context-free language. SIAM J. Comput. **12**(4) (1983) 645–655
19. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In Gupta, R., Amarasinghe, S.P., eds.: PLDI, ACM (2008) 206–215
20. Maurer, P.M.: The design and implementation of a grammar-based data generator. Softw., Pract. Exper. **22**(3) (1992) 223–244
21. Purdom, P.: A sentence generator for testing parsers. BIT **12**(3) (1972) 366–375
22. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York, NY, USA, ACM Press (September 2007)

23. Coppit, D., Lian, J.: yagg: an easy-to-use generator for structured test inputs. In Redmiles, D.F., Ellman, T., Zisman, A., eds.: ASE, ACM (2005) 356–359
24. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In Uyar, M.Ü., Duale, A.Y., Fecko, M.A., eds.: TestCom. Volume 3964 of Lecture Notes in Computer Science., Springer (2006) 19–38
25. Bardin, S., Finkel, A., Leroux, J., Petrucci, L.: Fast: acceleration from theory to practice. STTT **10**(5) (2008) 401–424
26. Bojanczyk, M., Segoufin, L.: Tree languages defined in first-order logic with one quantifier alternation. In Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I., eds.: ICALP (2). Volume 5126 of Lecture Notes in Computer Science., Springer (2008) 233–245
27. Pnueli, A.: The temporal logic of programs. In: FOCS, IEEE (1977) 46–57
28. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In Berry, G., Comon, H., Finkel, A., eds.: Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01). Volume 2102 of Lecture Notes in Computer Science., Paris, France, Springer (July 2001) 53–65
29. Somenzi, F., Bloem, R.: Efficient büchi automata from ltl formulae. In Emerson, E.A., Sistla, A.P., eds.: CAV. Volume 1855 of Lecture Notes in Computer Science., Springer (2000) 248–263
30. Holzmann, G.J.: The model checker spin. IEEE Trans. Software Eng. **23**(5) (1997) 279–295
31. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering tobias combinatorial test suites. In Wermelinger, M., Margaria, T., eds.: FASE. Volume 2984 of Lecture Notes in Computer Science., Springer (2004) 281–294
32. Dadeau, F., Kermadec, A.D., Tissot, R.: Combining scenario- and model-based testing to ensure posix compliance. In Börger, E., Butler, M.J., Bowen, J.P., Boca, P., eds.: ABZ. Volume 5238 of Lecture Notes in Computer Science., Springer (2008) 153–166
33. Aiguier, M., Arnould, A., Gall, P.L., Longuet, D.: Test selection criteria for quantifier-free first-order specifications. In Arbab, F., Sirjani, M., eds.: FSEN. Volume 4767 of Lecture Notes in Computer Science., Springer (2007) 144–159
34. Aiguier, M., Arnould, A., Boin, C., Gall, P.L., Marre, B.: Testing from algebraic specifications: Test data set selection by unfolding axioms. In Grieskamp, W., Weise, C., eds.: FATES. Volume 3997 of Lecture Notes in Computer Science., Springer (2005) 203–217
35. Gaudel, M.C., Gall, P.L.: Testing data types implementations from algebraic specifications. In Hierons, R.M., Bowen, J.P., Harman, M., eds.: Formal Methods and Testing. Volume 4949 of Lecture Notes in Computer Science., Springer (2008) 209–239
36. Jard, C., Jéron, T.: Tgv: theory, principles and algorithms. STTT **7**(4) (2005) 297–315
37. Tretmans, J.: Testing concurrent systems: A formal approach. In Baeten, J.C.M., Mauw, S., eds.: CONCUR. Volume 1664 of Lecture Notes in Computer Science., Springer (1999) 46–65
38. Stirewalt, R.E.K., Egyed, A., Fischer, B., eds.: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA. In Stirewalt, R.E.K., Egyed, A., Fischer, B., eds.: ASE, ACM (2007)